1. Calcolare la complessità $T(n)$ del seguente algoritmo MYSTERY1:

Algorithm 1: MYSTERY1(INT n)

```
if n ≤ 1 then
    return 123
else
    k ←MYSTERY2(n/2)+MYSTERY1(n/3)
    return k+MYSTERY1(n/3)
end

/* funzione ausiliaria                                          */
function MYSTERY2(INT m)
if m = 0 then
    return 321
else
    return 2×MYSTERY2(m/4)−MYSTERY2(m/4)
end
```

Soluzione L'algoritmo MYSTERY1 invoca MYSTERY2. Iniziamo quindi a calcolare il tempo di calcolo $T'(m)$ di MYSTERY2. La funzione MYSTERY2 è ricorsiva con costo $T'(m)$ che soddisfa la seguente relazione di ricorrenza:

$$T'(m) = \begin{cases} 1 & \text{se } n \leq 1 \\ 2T'(m/4) + 1 & \text{altrimenti} \end{cases}$$

Applicando il master theorem otteniamo: $\alpha = \frac{\log a}{\log b} = \frac{\log 2}{\log 4} = \frac{1}{2}$ e $\beta = 0$. Avendo $\alpha > \beta$, applichiamo il primo caso del master theorem ottenendo $T'(m) = \Theta(m^{\alpha}) = \Theta(m^{\frac{1}{2}})$. Ora consideriamo il costo $T(n)$ della funzione ricorsiva MYSTERY1 che soddisfa la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} 1 & \text{se } n \leq 1 \\ 2T(n/3) + (n/2)^{\frac{1}{2}} & \text{altrimenti} \end{cases}$$

Ma $(n/2)^{\frac{1}{2}}$ coincide con $(1/2)^{\frac{1}{2}} \times n^{\frac{1}{2}} = cn^{\frac{1}{2}}$, con c costante. Possiamo quindi applicare il master theorem ottenendo: $\alpha = \frac{\log a}{\log b} = \frac{\log 2}{\log 3} = \frac{1}{\log 3}$ e $\beta = \frac{1}{2}$. Avendo $\alpha > \beta$, applichiamo il primo caso del master theorem ottenendo $T(n) = \Theta(n^{\alpha}) = \Theta(n^{\frac{1}{\log 3}})$.

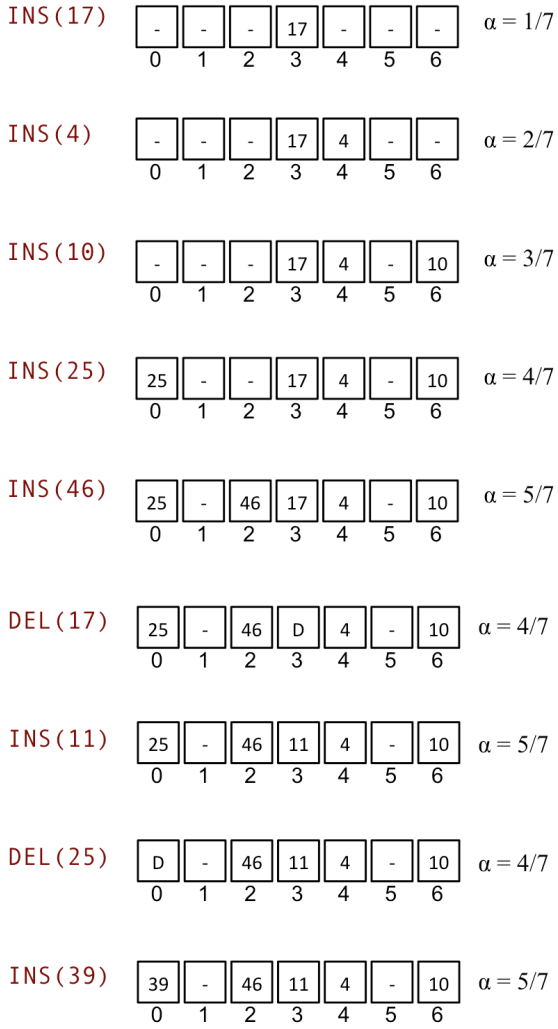
2. Si consideri una tabella hash di dimensione $m = 7$ inizialmente vuota. Le collisioni sono gestite mediante indirizzamento aperto usando la seguente funzione hash $h(k)$:

   $h(k) = (h'(k) + 3i) \mod m$

   $h'(k) = k \mod m$

   Si mostri il contenuto della tabella e il fattore di carico dopo ognuna delle seguenti operazioni, eseguite in ordine: INS(17), INS(4), INS(10), INS(25), INS(46), DEL(17), INS(11), DEL(25), INS(39).

   Soluzione La seguente immagine mostra la tabella hash e il fattore di carico (n/m) dopo ogni operazione:



3. Si consideri il seguente balletto medievale ballato da n maschi e n femmine. Inizialmente tutti i ballerini si collocano lungo una linea retta a distanza di un passo l'uno dall'altro. In una fase iniziale, le femmine, una alla volta, si muovono per raggiungere un maschio con cui formeranno una coppia per il resto del balletto. Le femmine si muovono a tempo, e ad ogni battuta della musica, la femmina che si sta muovendo fa un passo spostandosi a proprio piacimento verso destra oppure sinistra. Una volta raggiunto il compagno con cui formerà la coppia, alla battuta successiva un'altra femmina inizierà i propri spostamenti. Data una disposizione iniziale dei ballerini, bisogna capire quanto tempo sarà necessario per completare la formazione delle coppie. Il tempo di formazione di una coppia coincide con il numero complessivo di passi che devono effettuare le femmine per completare la formazione delle coppie. Progettare quindi un algoritmo che dato un vettore di booleani $B[1..2n]$ che indica le posizioni iniziali dei ballerini ($B[i] = true$ indica che in posizione i

è presente una femmina, altrimenti se $B[i] = \mathit{false}$ in posizione i è presente un maschio), restituisce un intero che rappresenta la durata minima possibile per la formazione di tutte le n coppie (ovvero, il numero minimo complessivo di passi che le femmine devono fare per formare le coppie).

Soluzione Il problema può essere risolto tramite un approccio greedy, abbinando l'i-esima femmina con l'i-esimo maschio. Per calcolare il tempo per completare la formazione delle coppie, si sommano i passi che le femmine devono fare per raggiungere il proprio maschio abbinato. Gli abbinamenti vengono calcolati inserendo gli indici delle posizioni delle ballerine, e dei ballerini, in due distinte code FIFO. Successivamente si estraggono gli indici abbinati e si calcola la distanza considerando il valore assoluto della differenza degli indici.

Algorithm 2: BALLETTO(BOOL $B[1..2n]$)→INT

```
QUEUE maschi ← new QUEUE()
QUEUE femmine ← new QUEUE()
for j ← 1 to 2n do
    if B[j] then
        femmine.enqueue(j)
    else
        maschi.enqueue(j)
    end
end
tot ← 0
for j ← 1 to n do
    tot ← tot+ABS(femmine.dequeue() − maschi.dequeue())
end
return tot
```

Il costo computazionale di tale algoritmo risulta essere $T(n) = \Theta(n)$, assumendo che le operazioni di *queue* ed *enqueue* abbiano costo costante, visto che tutte le operazioni hanno tempo costante e vista la presenza di due **for** che eseguono entrambi $2n$ cicli.

4. Progettare un algoritmo che, dato un grafo orientato $G = (V, E)$ ed un vertice $v \in V$, restituisce il numero di vertici di un ciclo di lunghezza minima a cui v appartiene (restituisce ∞ se v non appartiene ad alcun ciclo).

Soluzione È possibile usare una visita in ampiezza partendo dal vertice v. Infatti, in questo modo si visitano gli archi in ordine di distanza non decrescente a partire da v. Appena si visita un nodo adiacente a v, si può concludere che la lunghezza del ciclo minimo coincide con la distanza di tale nodo più 1. Se si termina la visita senza trovare alcun ciclo, tale ciclo non esiste e si può restituire ∞.

Il costo computazionale dell'algoritmo coincide con quello della visita in ampiezza, ovvero $O(m + n)$ con m numero di archi ed n numero di vertici del grafo. Si noti che a differenza della visita in ampiezza, nel caso in cui esista il ciclo, tale algoritmo può terminare prima di aver visitato l'intero grafo. Ma nel caso pessimo, che si verifica in assenza di tale ciclo, si rende necessario visitare l'intero grafo.

Cont.

Algorithm 3: CicloMinimo(Graph $G = (V, E)$, Vertex v) $\rightarrow$ Int

```
Queue q ← new Queue()
for x ∈ V do
    x.mark ← false
    x.dist ← ∞
end
v.mark ← true
v.dist ← 0
q.enqueue(v)
while not q.isEmpty() do
    u ← q.dequeue()
    if v ∈ u.adjacents() then
        return u.dist + 1
    else
        for w ∈ u.adjacents() do
            if not w.mark then
                w.mark ← true
                w.dist ← u.dist + 1
                q.enqueue(w)
            end
        end
    end
end
return ∞
```

The End.